High-level Synthesis for Semi-global Matching: Is the juice worth the squeeze?

High-level Synthesis for Semi-global Matching: Is the juice worth the squeeze?

Affaq Qamar, *Member, IEEE,* Fahad Bin Muslim, *Student Member, IEEE,* Francesco Gregoretti, *Member, IEEE,* Luciano Lavagno, *Senior Member, IEEE* and Mihai Teodor Lazarescu, *Member, IEEE*

***Abstract*—High-level Synthesis (HLS) based design methodologies are extremely viable for industries that are sensitive to production costs. In order to have competitive advantage, the ability to have several different implementations of the same algorithm satisfying a diverse range of resolution, cost and performance constraints is highly desirable. In this article, we present multiple hardware implementations of the Semi-global Matching (SGM) algorithm which is used in stereo vision systems e.g. for automotive applications. The hardware platform considered in this work is a Xilinx® Zynq™ System-on-Chip. A performance comparison of both HLS-based design as well as a manual RTL design in terms of quality of results (QoR), flexibility and design time is also presented. SGM mainly includes a sequence of three processing steps i.e. the "cost cube calculation" followed by the "path cost computation" and finally the "disparity approximation and minimization". The path cost processor further performs a pixel-wise processing of the cost cube data along eight distinct path orientations. The baseline algorithmic model usually called the "golden" model utilizes considerably large arrays, that are required to be mapped to an external DRAM and brought into the on-chip RAM when required. This necessitates adding both the memory transfer loops as well as insertion of calls to the AXI transactors for accessing the DRAM through the on-chip DDR slave. Furthermore, the initial algorithm (typically single-threaded) must be parallelized to fully exploit the concurrency offered by the target hardware platform. The design space exploration was thus performed by making several considerably different micro-architectural choices. Eventually, we were able to obtain an implementation comparable to the manual RTL design. Both manual RTL as well as the HLS designs achieved the target real-time performance of 30 fps for the image resolution of 640x480 with a disparity depth of 128 pixels per frame.**

***Index Terms*—High-level Synthesis, FPGA, RTL, Semi-global Matching, DRAM, Design Space Exploration.**

I. Introduction

SYSTEM-ON-CHIP (SoC) designs are becoming increasingly heterogeneous as they combine multicore architectures with a variety of hardware accelerators to carry out dedicated computational tasks. These hardware accelerators offer several orders of magnitude higher power and timing efficiency than a corresponding software implementation [1]. However, the presence of accelerators aggravates the complexity of SoC design. With the continuous advancements in technology, the complexity of electronic designs now has a profound effect on the overall cost, performance, and power consumption of the modern electronic systems.

A. Qamar is with the Department of Electrical Engineering, Abasyn University, Peshawar, KP, 25000 Pakistan e-mail: affaq.qamar@abasyn.edu.pk

F. B. Muslim, F. Gregoretti, L. Lavagno and M. T. Lazarescu are with the Department of Electronics and Telecommunications, Politecnico di Torino, Italy.

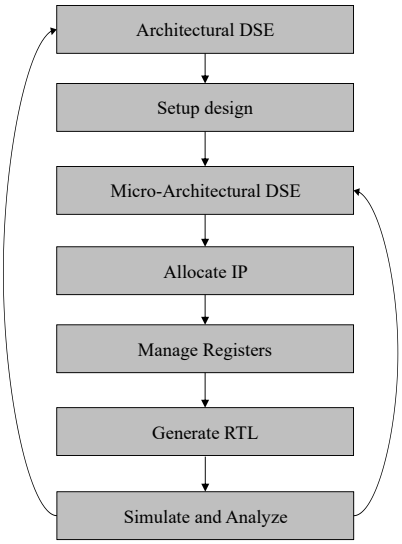

Fig. 1. General high-level synthesis flow.

As far as behavioral description for the hardware design is concerned, the abstraction level is rising from RTL to algorithmic untimed or transaction-based, followed by an automated high-level synthesis (HLS) flow [2]. Model-based Design (MBD) is a methodology that starts from an abstract, implementation-independent model that is functionally verified and algorithmically optimized. It then maps the model to several optimized candidate implementations, eventually choosing the one that best meets the market requirements. In the context of hardware design, MBD uses high-level synthesis for this mapping.

HLS takes as input the model-based description of the design, specified in some high-level language such as C, C++, SystemC or Simulink, and synthesizes it to generate RTL, as depicted in Fig. 1. By elaborating different sets of constraints, HLS tools allow designers to evaluate multiple implementation alternatives, a process known as Design Space Exploration (DSE) [3], [4]. HLS enables the description of a digital design at a higher level of abstraction accompanied by different design constraints by using control and data flow graphs (CDFG). CDFG scheduling and binding with respect to these constraints enable us to explore the design space more rapidly thereby improving the Quality of Results (QoR) compared to the manually-coded RTL design.

Design space exploration with HLS is much broader and easier than what is possible with logic synthesis alone, since

the former can be achieved by simply changing HLS tool directives, while the latter usually requires one to manually change a detailed hardware description expressed in the form of Verilog or VHDL code. Such hardware descriptions at a lower abstraction level are often attempted only for a limited number of architectural options, because of the corresponding larger design and verification times [5]. Owing to its improved design re-use, reduced simulation run-time and a broader design space exploration, HLS considerably reduces the time to market and improves the coding productivity. Additionally, high design productivity necessitates that complex SoCs use a higher percentage of reused components [6]. This in turn requires soft IP components, that are designed only once using some high-level languages and are implemented at various instances in order to meet various design requirements [7].

A C-based algorithmic model is usually available as a reference "golden" model to both HLS and RTL designers. This model however, needs several modifications before it can be synthesized automatically via HLS. This is particularly true when the reference model (typically single-threaded) lacks sufficient parallelism, thus making it very difficult to meet the design constraints. This hence, requires putting in some effort to manually parallelize the code by splitting it into multiple synchronized threads communicating through e.g. FIFOs or ping-pong buffers. This manual effort however, is still considerably smaller than the extensive coding effort required for direct RTL implementation.

In addition to concurrency, another aspect worth considering while gauging the manual effort is the memory access optimization, which usually is the bottleneck, particularly for image and video processing algorithms. This becomes even more crucial when the design implementation requires accesses to external dynamic random access memory (DRAM). This requires writing custom-built memory transfers to on-chip SRAM buffers which replace the caches that are being used in the respective software implementation of the algorithm.

A. Problem Statement

This article addresses some important issues related to high-level synthesis and system-level design in general. Synthesizing a design from an algorithmic (also called system-level) model using an automated HLS flow provides efficient implementation in terms of area, performance and power with respect to its software counterpart. However, the high-level code requires to be modified considerably with regards to a pure simulation model in order to ensure an implementation that is comparable to a highly efficient (and therefore very rigid) manual RTL design written in hardware description languages such as Verilog or VHDL. In particular, it must consist of a very sophisticated mechanism to ensure a fine-grained management of data and computation.

This apparently deviates somewhat from the stated goal of MBD that is "model once, run anywhere". However, it still follows broadly the MBD guidelines, because these optimizations are beneficial for all hardware implementations derived from HLS, and simply maximize the size of the design space that can be explored, while simultaneously optimizing the QoR. Typically these modifications include: (1) increasing the level of explicit parallelism in the model, since its automated extraction from a sequential model is almost impossible for a tool, and (2) restructuring the memory accesses to better exploit their locality, since in hardware there is no cache to provide the illusion of a very fast and huge memory.

Video processing algorithms are widely used in the field of machine vision applied e.g. to the automotive and surveillance domains. An efficient vision system in an automobile is a promising technological solution to replace human interaction during driving. In order to cater to a wide range of vehicles in a cost-effective manner, the ability to achieve several varieties of a single design offering a diverse range of cost and performance, would lead to a great competitive advantage.

B. Contribution

The scope of the paper covers the manual transformations needed in order to get an efficient hardware implementation from a high-level code. This article targets the design space exploration of a Stereo Vision System (SVS), which is a reasonably complex design (i.e. it can fit over a high-end FPGA). We intend to explore the application of HLS to a complex design involving memory-intensive operations. The test case under consideration is the FPGA implementation of a Semi-global Matching (SGM) algorithm which is being employed in a Stereo Vision System. SGM finds broad range of applications in the automotive domain, e.g. in assisted driving applications, and it needs to be executed in real-time while adhering to strict cost and power constraints. SVS uses a set of 2D images taken by two cameras separated by some distance to construct a 3D image frame, as illustrated in Fig. 2.

The process is based on a disparity estimation technique using SGM. To cater to a broader range of vehicles in a cost-effective manner, the ability to have multiple implementations of a design satisfying various cost/performance constraints helps achieving a tremendous competitive advantage. The manual RTL for this design was already implemented for an automotive company in the context of a European Research Council (ERC) Sensor for 3D Vision (3DV#297463) Proof of Concept grant [8]. In the present work, we explore an alternate approach to achieve comparable results with much lower design effort and a higher degree of re-usability. The latter is particularly important for the automobile industry which is highly sensitive to cost. This industry requires various rapid and diverse implementations of the same algorithm satisfying various resolution, cost and performance metrics targeting different market segments. A comparsion of the two flows in terms of both flexibility and QoR will be presented in this article.

C. Paper Organization

The rest of the article is organized as follows. Section II presents a brief overview of the SGM working principle and its algorithmic background. Some relevant state-of-the-art work is presented in Section III. Section IV discusses the algorithmic refinements needed at the pre-synthesis stage

followed by the hardware-software partitioning. Three different hardware architectures and their implementations are presented in Section V. It starts with the memory access optimizations and then presents a single threaded sequential architecture. Section V also highlights the introduction of parallelism using manual effort as well as tool-based micro-architectural decisions. Section VI discusses the design space exploration setup and design analysis using HLS and compares its results with current state-of-the-art. The work is concluded in Section VII.

II. Stereo Vision Systems

Stereo vision systems have multiple cameras fixed over a common platform, all capturing the same scene. Slight differences in the points of view of the cameras produce small displacements of the various objects in the camera images, in exactly the same manner as in case of human eyes. These displacements can be used to obtain a three-dimensional model of the framed scene. This can be done by adding the displacement of each object to its two-dimensional position, estimated from each image. The distance computation becomes extremely simple if the displacements are known, as it will only depend on the baseline between the cameras. The displacement calculation however, is not straight forward since it includes an extremely complicated comparison between the various camera images [9].

In this work, we consider an automotive application that detects obstacles and their distance from a vehicle using SVS. The system consists of two cameras that are assumed to be mounted in front of a car, taking numerous shots that need to be analyzed in real-time. The sensors in the cameras are positioned such that their longer side is aligned with the baseline between the two cameras. This ensures that the displacements occur in a single dimension only hence, making the vision algorithm considerably simpler. Considering the baseline to be horizontal, then the two images are termed as left and right images respectively. The system gives output in the form of an image, which contains, for each pixel, its estimated distance from the cameras. This is followed by another algorithm for detecting the presence of an obstacle. Furthermore, in the case of an obstacle being detected, automatic intervention is ensured as well [9]. Among numerous potential stereo vision approaches, SGM algorithm is selected here due to its greater robustness and regularity [10]. These properties are important as they can be exploited for an efficient hardware implementation and design space exploration as well.

A. Semi-Global Matching (SGM)

One way to determine a three dimensional model of a scene is by considering a pair of images taken at the same time and calculating the displacement of all pixels in those images. This problem is typically called "image registration", and has applications in several domains e.g. remote sensing, medical imaging and computer vision, to name a few. Several approaches for image registration are surveyed in [11]. Semi-global matching however, provides the best known approach for image registration [10], [12].

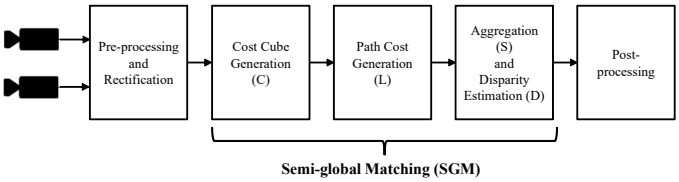

Fig. 2. Semi-global Matching based stereo vision system.

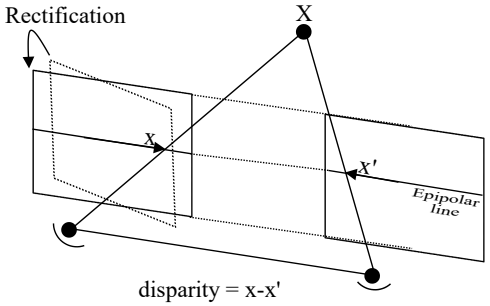

Fig. 3. Illustration of disparity in a stereo vision system.

The main blocks of an SVS system are shown in Fig. 2. The two images, each with WxH pixels, need to be pre-processed for noise reduction. Rectification, thereafter, is used to compensate the effects of camera distortion and sensor misalignment. A Look-up table (LUT) is used to perform rectification, which gives, for every pixel in the rectified image, the coordinates of the corresponding pixel in the original input image. An external DRAM is used to store the rectified pixels for census computation. For a square $(n \times n)$ window around a pixel, census represents a string of n^2 bits. Each of these bits equals 1 if intensity of the corresponding pixel in the window is higher than that of the center pixel while it equals 0 otherwise [9]. SGM uses the census transforms C_L and C_R corresponding to the left and right images respectively for computing the cost cube over disparity of depth d. The disparity is the difference in locations of the projection points in both the images as depicted in Fig. 3, where the rectification is only illustrated for the left projection plane. These steps are described next.

1) Cost Cube Generator: It computes the cost cube using the Hamming distance between pixels over the 128 disparity levels by scanning the left and right images. The corresponding mathematical equation is given by (1):

$$C(\mathbf{p}, d) = H(C_L(x + d, y), C_R(x, y)) \tag{1}$$

Where $\mathbf{p} = [x, y]^T$ represents the location of a pixel in the base image.

2) Path Costs Processor: This represents the major computation task corresponding to SGM. It aggregates the path costs $L_r(\mathbf{p}, d)$ along multiple independent paths in a recursive manner. Paths may be arbitrary, but are usually one-dimensional while following the main Cartesian axes and those at 45°, as shown in Fig. 4, which depicts eight separate paths $(P_1, P_2, ..., P_8)$ for a specific pixel $\mathbf{p}$. The cost L_r, for a path r for a specific pixel $\mathbf{p}$ is represented by (2).

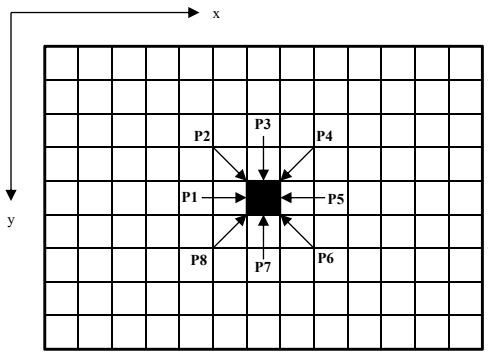

Fig. 4. Eight path orientation for path cost calculation.

$$\begin{aligned} L_r(\mathbf{p},d) = C(\mathbf{p},d) + \min[&L_r(\mathbf{p}-r,d),\\ &L_r(\mathbf{p}-r,d-1)+P_1,\\ &L_r(\mathbf{p}-r,d+1)+P_1,\\ &\min_i L_r(\mathbf{p}-r,i)+P_2]-\\ &\min_l L_r(\mathbf{p}-r,l) \end{aligned} \tag{2}$$

3) Aggregation and disparity estimation: The aggregated cost for each pixel is calculated by adding the path costs as presented in (3). The disparity is finally calculated by minimizing the aggregated costs as shown in (4).

$$S(\mathbf{p},d) = \sum_r L_r(\mathbf{p},d) \tag{3}$$

$$D(\mathbf{p}) = \min_d S(\mathbf{p},d) \tag{4}$$

In this work, we assume a VGA frame where; $W = 640$, $H = 480$, $n = 5$ and $d = 128$.

III. Related Work

A. HLS-based work

Most of the previous work on the SGM algorithm pertains with the RTL implementation of the SGM algorithm. One of the few exceptions is [9], where a sequential SystemC model of the SGM algorithm is implemented on a Xilinx Zynq 7020. The design space exploration results suggest that the code required huge arrays which were mapped to the external DRAM. Furthermore, the performance in terms of cycles per pixel (CPP) is affected not only by the frequent DRAM accesses but also by the sequential nature of the code. Thus, with this sort of implementation, the target of 30 fps with the image resolution of 480x640 for 128 disparity levels can not be achieved. Some architectural as well as algorithmic modifications are proposed in [13]. This work presents an integrated approach that combines memory partitioning and merging with data reuse and loop unrolling for optimizing memory organization for FPGA behavioral synthesis. The performance (cycles per pixel) is better than the one proposed in [9]. However, the size of the arrays implied by this FIFO-based strategy requires huge on-chip BRAM resources which are available only in the Xilinx Virtex-7 v2000T. This makes it a resource-dependent implementation which is not desirable.

The authors of [14] have tried to manage array reuse mainly through loop tiling, while in [15] they have suggested a powerful dependence distance approach that organizes the reused data in sets. In [16], buffer allocation reuse was optimized by a heuristic algorithm. The same authors also generated an on-chip reuse buffer in [17] by combining loop transformation and memory hierarchy allocation. All of the cases mentioned here involve sequential execution models, thereby enabling the sharing of the reuse buffer among all the arrays without any conflicts in access. The same techniques would not however, be suitable for loop unrolling, wherein access conflicts may arise due to limited number of ports of the physical RAMs as a result of concurrent data requests. Hence, direct combination of data reuse along with loop unrolling may not yield the expected improvement in throughput.

Memory allocation and binding guided by scheduling has been used in a lot of cases to avoid access conflicts. Moreover, traditional scheduling and binding algorithms that are used by many state-of-the-art HLS tools e.g. Vivado HLS from Xilinx [18], C-to-Silicon from Cadence [19], Catapult from Mentor Graphics [20] and Synphony from Synopsys [21] also do not cater to the needs of embedded designs involving complex image processing algorithms and stricter performance and area constraints [22]. This is because these tools are mainly designed to optimize scalar operations. The authors in [23] have used instruction-level macro-rescheduling and memory access-level micro-rescheduling to choose the best allocation and binding strategy. Although such methods can reduce the access conflicts and latency, yet the improvement in performance will be limited without some type of data layout optimization e.g. data reuse. A simple solution to reduce access conflicts can be to increase the number of memory ports. However, this leads to quadratic growth in complexity and area, which is inefficient and unrealistic [24].

An alternate solution to manage port constraints is to partition the memory into several banks with an acceptable overhead. The authors of [25] have designed a logical-to-physical mapping algorithm to break and pack memories into dual port RAM. The on-chip SRAM has been partitioned using an application-driven approach in [26], where frequently accessed data is mapped to smaller power-efficient memory units using application profiling. For reconfigurable architectures, the authors of [27] utilize memory distribution, replication and scalar replacement to map arrays of data to heterogeneous storage resources. The approach is used to combine a high-level specification with scheduling. A profiling-based approach that considers the partitioning of elements into data structures for behavior-level synthesis has been presented in [28], aiming to increase memory parallelism by data partitioning. Memory partitioning has been automated in [29] to achieve maximum throughput while using loop pipelining. All these methods need affine indices, while data reuse buffers are always updated in a circular manner to save buffer sizes, hence bringing modulo operations into the indices. None of the above mentioned strategies would work in such circumstances.

The authors of [30], [31] implemented an integer non-

linear programming model for data reuse and loop-level parallelization which solves the issue of access conflict by using memory and data duplication. This solution yields better performance, but causes an increase in the on-chip RAM and a redundancy in the data movement due to the process of memory duplication.

Loop unrolling for general purpose and embedded processors has been extensively studied in compilers [32], [33]. Several compiler optimizations and transformations e.g. sub-expression elimination, speculation, loop retiming and pipelining, and bit-level optimizations, have been explored and adapted to HLS flows in [34]. The area and performance impact of such transformations when mapping applications onto re-configurable processors has been studied recently in [29]. Loop unrolling has been addressed in articles such as [35] but the unroll factor had to be specified manually. Additionally, several commercial HLS tools e.g. SystemC Compiler, Xilinx Vivado HLS and CatapultC, also require the designer to either explicitly instruct the tool to completely unroll a particular loop, or explicitly specify an unroll factor for partial unrolling.

B. RTL-based work

An RTL implementation of SGM having a VGA image resolution i.e. 640x480 pixels running at 30 fps has been presented in [8]. The implementation platform considered in this work is the Xilinx® Zynq™ 7020 board, which yields a significant reduction in development effort as this board incorporates an FPGA, a dual-core ARM processor and multiple I/Os.

Some algorithmic extensions for power efficiency have been made to the SGM implementation in [36]. The resolution of the proposed implementation is 340x200 pixels, which is one fourth of our target, and it can achieve a frame rate of 27 fps.

A novel two-way parallelization-based architecture has been presented in [37] to obtain highly efficient computation. The systolic-array based architecture also ensures easy scalability in terms of frame rates and image resolutions. The hardware platform selected in this work is a Xilinx® Virtex-5 platform with a VGA image resolution and a frame rate of 30 fps.

An alternative SGM implementation on an Nvidia® Tesla C2050 Graphics Processing Unit (GPU) has been presented in [12]. Disparity estimation in the proposed implementation is performed at 27 fps for an image resolution of 1024x768 pixels with 128 disparity levels. The GPU performance in this case is quite exceptional but the implementation does not meet our application requirements in terms of size and power consumption and hence this approach is not considered.

In the present work, we have performed manual memory optimizations such as minimization of read/write operations, resolution of access conflict by design-centric address analysis and shrinking down of huge arrays into memory banks in order to avoid data dependencies. We also exploited the HLS tool capabilities to define micro-architecture by partially unrolling loops. This was combined with array splitting for the memory banks that were used in unrolled loops.

IV. Algorithmic Refinements of SGM

The system-level code of the SGM algorithm must undergo some refinements in order to achieve better QoR. Some of these transformations deal with datatype conversion, since for hardware the bit count is crucial both for the I/O ports as well as for the communication interfaces. The rest of the refinements target algorithmic and architectural modifications in order to achieve the area and performance constraints.

A. Pre-transformations of reference code

The algorithmic C/C++ based model, that is usually available as a reference for hardware implementation, needs considerable amount of modifications before it can undergo automated RTL synthesis. These modifications involve:

- converting global variable declarations to local declarations.
- converting floating point arithmetic to fixed-point.
- converting dynamic arrays to static arrays.

These transformations are necessary in order to achieve an efficiently *synthesizable model*. Afterwards, the information regarding the I/O ports and the synchronization mechanism required by the code to communicate with the camera sensors and the output memory are required. This is accomplished by using the SystemC as an HLS input language. SystemC is a C++ class library designed to effectively create a Transaction-level model (TLM) or cycle-accurate model of functionality, hardware architecture, and interfaces required for system-level designs [38]. It offers the necessary constructs required to build a system architecture model, including information regarding ports, timing information and concurrency etc that are absent in standard C++. A simplified graphic illustration of high-level code assembly into a SystemC wrapper is depicted in Fig. 5. The main advantage of this assembly stems from the distinction between the behavioral model of the system and the data communication, that this embedding achieves. The source code is connected with the rest of the system through the port interfaces and the communication channels.

Path aggregation results in the production of a huge amount of data for each frame (data width x VGA frame size x number of disparity levels). Since the target hardware platform is an FPGA with limited on-board RAM, such a huge array needs to be mapped to an external DRAM. AXI bus communication can be used to access the external DRAM in Xilinx 7 series FPGAs. As illustrated in Fig. 6, the AXI protocol uses a master transactor in order to transform the memory access requests made by the SGM module into AXI signals. The AXI slave transactor; which is available on the 7 series FPGA fabric in the form of memory interface solution, is used to interpret the data read/write requests to DRAM, to synchronize and reorder them so as to maximize the throughput and translate them in to actual signals to be sent to the DRAM [13].

B. Software vs. hardware

For HLS to work efficiently for real-time applications, some refinements must be made to the algorithm in order to obtain a desirable HW implementation. These modifications mostly

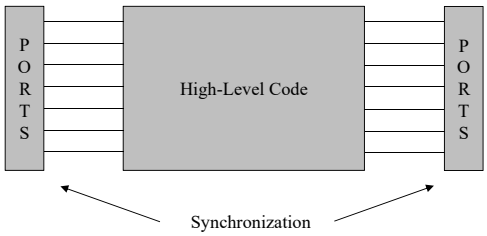

Fig. 5. SystemC wrapper around the high-level code for HLS.

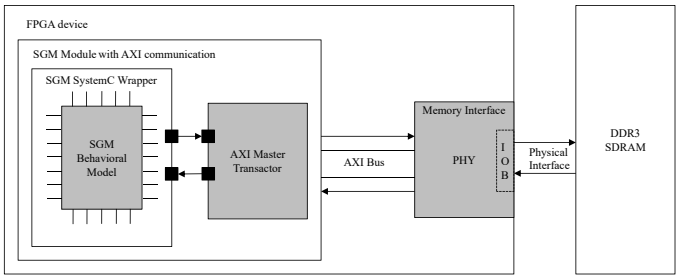

Fig. 6. Illustration of SGM Module with AXI interface for external DRAM access.

stem from the fact that while memory is cheap and parallelism is limited in software, in HW fast memory is expensive but a high level of parallelism can be available. As a matter of fact, most of the architectural exploration in an image processing application such as SGM tries to *match the rates of memory read/write and those of data computation*. When that is achieved, the Pareto-optimal cost/performance design space exploration points are obtained.

Fig. 7 displays the modifications that were made to the non-hardware specific code in order to achieve a desirable hardware implementation. The original software code calculates the cost cube for each pixel once thereby storing it in a very large memory as shown in (Fig. 7(a)). On the contrary, the cost cube computation of each pixel in the modified code takes place on the fly, while calculating the path costs of the specific pixel, and is stored into a FIFO register of width 128 as shown in (Fig. 7(b)). The result of the computation from the FIFO is updated by the cost cube of the next pixel, since the scan handles one pixel at a time.

C. Hardware-software partitioning

The present work focuses on the DSE of the SGM algorithm. For this, the cost cube and path cost calculation along with path aggregation and disparity estimation blocks are targeted for hardware implementation. We used the Xilinx Zynq^{TM} 7020 SoC which is equipped with both an FPGA Programmable Logic (PL) and a Programmable System (PS) with a dual-core ARM Cortex^{TM}-A9 CPU in the same physical packaging. On the other hand, the other key constituent blocks for stereo reconstruction, namely the rectification and census, as discussed in Section II, are kept as software implementation because they are not as performance critical. The intermediate results from the census transform of each VGA frame are stored in the external DRAM memory. The values of each pixel are then pre-fetched into local buffers and fed to the cost cube computation block. The implementation takes into account the throughput of the SGM block, assuming that the required parts of the census images are already available in the local buffers.

V. Hardware Architectures of the SGM Algorithm and High-Level Synthesis

In Section IV we discussed refinements at the algorithmic-level. We performed design space exploration of three different hardware architectures of the SGM algorithm based on manually defined parallelism, along with the tool-assisted coarse-grained micro-architectural decisions, in order to get various HW implementations covering a wide range of the area performance curve. This section discusses the manual transformations of the high-level code, leading to three different hardware architectures with different degrees of parallelism and memory utilization. Moreover, we also discuss the tool-assisted micro-architectural choices.

A. Micro-architectural decisions

The system-level test bench written in SystemC is used both for the functional verification of the high-level code as well as for the performance analysis of the resulting RTL, post-scheduling. For design space exploration, different micro-architectural decisions give several different RTL implementations each offering different area vs. performance trade-offs. The general flow to carry out HLS is illustrated in Fig. 1. The micro-architectural decisions involve loop, function and array implementation. Loop unrolling creates N copies of the loop body. The function calls inside the high-level code can be inlined or assigned to some specific IP block, while the arrays can be mapped to register files, block RAM (BRAM) or any vendor-supplied memory in case of ASIC implementation. Once such choices are made, the scheduling and binding step maps the functional blocks to the available resources fulfilling the latency constraints.

B. Array Optimizations

Memory accesses usually form bottlenecks in achieving higher performance optimization particularly for digital signal, image and video processing algorithms. Thus, they need to be optimized very carefully [13]. This is especially true, when the primary and intermediate input/output design data is too large to fit in the on-chip SRAM and an external DRAM is required. Customized memory transfers are required to be written in this case. Before the micro-architecture is defined, it is a good idea to move as many memory read/write operations as possible, outside the body of the loop [39]. Occasionally the HLS tool can do this, but in many cases, the tool needs to be guided explicitly to perform this optimization.

To emphasize on this point, consider the loop (for path cost accumulation) as given in Fig. 8, which is called numerous times and it accounts for around 25% of the overall computation. The loop contains two read operations for the array named pathcost[i]. If we unroll this loop 256 times while this array is assigned to an external memory e.g. DDR3, then the

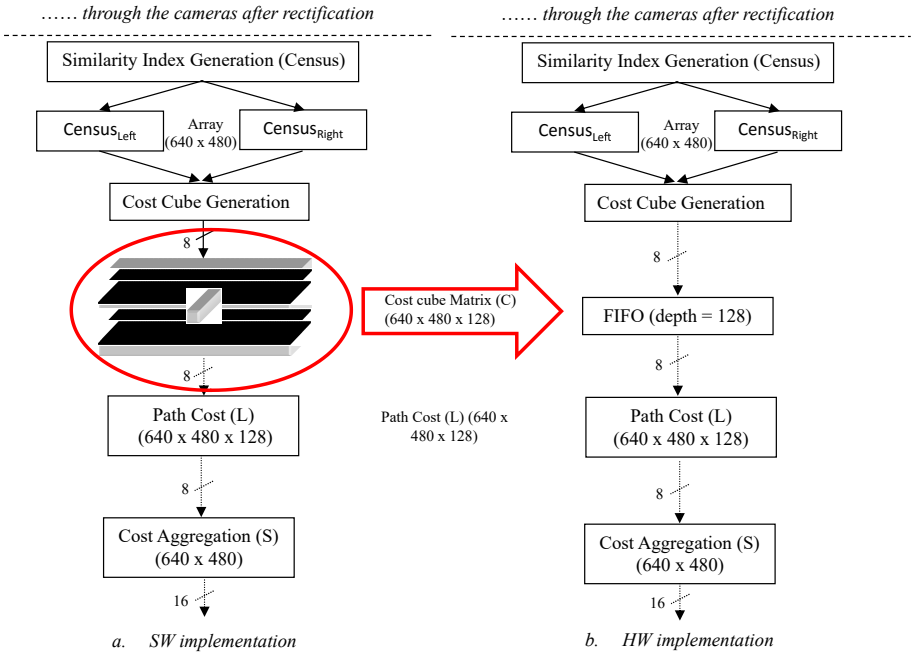

Fig. 7. Algorithmic refinement made at system-level for hardware implementation.

```
for(int i = 0; i < DISPARITY; i++)
{
……
uint8_t min_path = min(pathcost[i], next, previous);
……
previous = pathcost[i];
……
}
```

Fig. 8. Illustration of un-optimized read operation.

```
for(int i = 0; i < DISPARITY; i++)
{
……
uint8_t temp_path = pathcost[i];
uint8_t min_path = min(temp_path, next, previous);
……
previous = temp_path;
……
}
```

Fig. 9. Illustration of optimized read operations.

same memory location would need to be read from, two times per iteration of the loop i.e. 512 reads for the complete loop. Alternately this read operation can be executed as presented in Fig. 9, where the array is read once into a variable. This will lead to savings in read operations thereby improving both the resource utilization and the overall design throughput.

Table I indicates the optimizations made to the memory accesses, corresponding to the three main computational blocks of SGM. The former function is inlined to be used by all the processes, thus reading three successive on-chip BRAM locations and then writing back to it. Such reads can be joined together as bursts to achieve better performance while utilizing buffers, hence resulting in memory bandwidth saving. Both the latter functions (accessing much slower external DRAM) can be optimized in the same way to improve the overall QoR.

C. Single threaded sequential code

After describing the manually guided memory access optimizations, we are ready to discuss the impact of parallelism to reduce design latency, thereby improving the overall performance of the design. The performance measure is the CPP count, which is the total number of clock cycles taken by the SGM hardware to compute the disparity value of the full image, divided by the number of pixels. The single threaded code performs sequential computations similar to the original reference code. The only difference is that it underwent the synthesizability changes discussed in Section IV-A and the memory access optimizations described in Section V-B. The computation begins with the top left pixel, after omitting the first two pixels, and computes the cost cube (C) of all the pixels. This is because the census transform cannot be applied to the boundary pixels since it requires a 5x5 pixel window. The cost cube computation is performed only once and is used by all the path cost (L) calculation steps in their respective path orientation. For the first pixel, the path cost is simply the minimum between the cost cube and the maximum disparity value, which in our case is 128. The path costs of all the path orientations are aggregated alongside. Once the path costs are computed in all eight orientations, the disparity estimation is performed, as illustrated in Fig. 10. It is worth mentioning here that all the intermediate and final results in the form of disparity are stored into arrays. These arrays need to be mapped onto a larger, but comparably slower, Double Data Rate (DDR) memory (faster on-chip BRAMs may be used for

TABLE I
OPTIMIZATIONS APPLIED TO MEMORY ACCESSES

Function	Mem Type	Occurence	Iterations	Pre-optimization R/W Operations	Mem Access	Post-Optimization R/W Operations	Mem Access Optimization
Path Accumulation	SRAM	8	128	4	4096	2	2048
Aggregation	DRAM	8	128	4	4096	2	2048
Disparity Estimation	DRAM	1	128	5	640	2	320

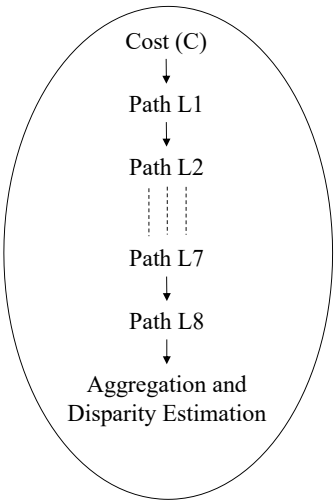

Fig. 10. Illustration of Single-threaded sequential architecture.

smaller arrays). As a result, a single threaded, sequential code would not be able to meet the performance constraints. This sequential architecture also has a very small BRAM usage, thus making it very inefficient also from the view point of resource utilization.

It should be noted that only the innermost loops (with the highest number of computations) are considered for analysis of various loop implementation options. The most obvious options for trade-off analysis, obtained as a result of profiling the initial high-level code, were the loops performing the path accumulation, and the loop performing Cost Cube generation.

When performing tool-assisted micro-architectural decisions for the sequential single-threaded architecture, we inline all the functions. Two types of loop implementations are tried for the cost cube and path cost loops. As an initial case, all the loops are resolved by adding sufficient *wait()* statements in the loop bodies to make them sequential and also accommodating all the BRAM access requirements of the algorithm. This is termed as "loop breaking". Xilinx supplies memory models for both BRAM (on-chip memory) and DDR (external IP). As a second variant, complete unrolling is performed on the most time-consuming loops so as to increase the concurrency (at a small expense of the area cost). All the other loops are resolved by breaking them as explained above. The arrays used inside the loops are flattened into registers, to obtain a smaller and faster implementation. The rest of the larger arrays are still mapped to on-chip BRAMs.

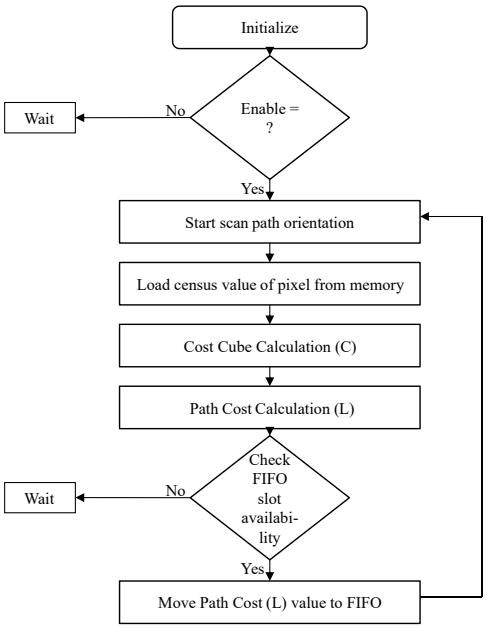

Fig. 11. Algorithmic description of one of the parallel SGM threads.

D. FIFO-based concurrent architecture

To improve the overall design throughput, the original single-threaded model needs to be split into multiple synchronized threads. The verification of such a transformation is still much simpler and faster than the huge manual effort required for the direct RTL implementation of the design [6].

The path cost calculation process is completely independent from one direction to the other and hence it is pertinent to divide the path cost calculation operation into eight different parallel processes. Nevertheless, as these threads access shared memory, either they need to be properly scheduled in order to prevent race conditions or local copies must be created in order to optimize the performance at the cost of additional BRAM resources. In our case, we followed the former strategy and introduced another concurrent process which performs arbitration and path cost accumulation. The computations carried out in each parallel process for path cost computation are depicted as a flow chart in Fig. 11.

The cost cube corresponding to each pixel along the 128 disparity levels is computed on the fly and is saved in a temporary memory buffer while moving along the direction of a single path. This implies that for each pixel, the cost cube is computed eight times, but this calculation replication can dramatically reduce the memory access cost. Without this, the cost cube values corresponding to the whole frame would

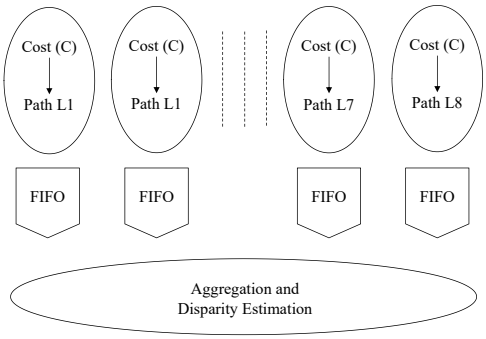

Fig. 12. Illustration of the SGM FIFO-based multi-threaded (nine) concurrent architecture.

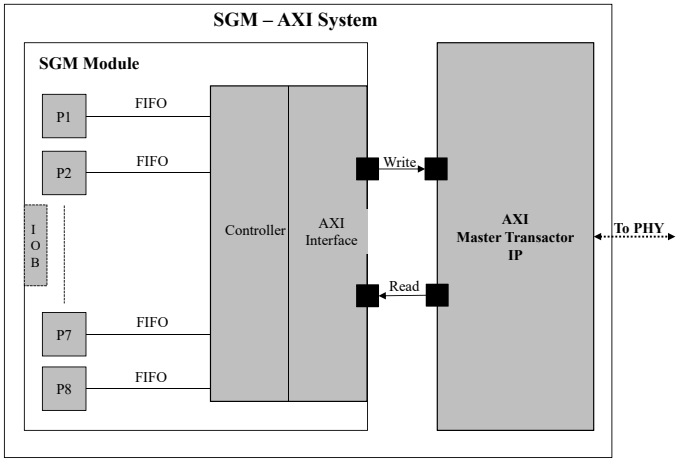

Fig. 13. Parallel SGM processes communication architecture via AXI interface.

need to be saved in an external DRAM, thereby causing a significant increase in expensive DRAM accesses. The path cost is then calculated and the result is stored into a dedicated FIFO, as illustrated in Fig. 12. The results from the FIFO channel are thereafter written into the external DRAM via AXI communication.

The communication architecture between the various processes of the SGM module and the AXI master transactor is depicted in Fig. 13. The controller enables all the SGM processes to begin the path computation. The controller also does the path costs aggregation by calling a method to read the older value from the DRAM, once the data is available at the FIFO, thereby accumulating it (with saturation) and writing it back. Once all the calculations for an entire frame are completed, the path processes are deactivated by the controller. Data is then read back from the DRAM for disparity computation of each pixel.

The FIFO-based architecture enhances the performance by a factor of ten with respect to the single-threaded implementation discussed before. However, its shortcoming is the large size of the eight FIFOs which are to be mapped to the on-chip BRAMs. Each of the path accumulation processes stores the path cost results into its specific FIFO, whereas the arbitration operation of the controller reads the path costs back from the FIFO channels corresponding to all the eight paths, which

```
Parallel Thread 1 - UNROLL_INSTANCE_1:
for(int i = 0; d < even_iterations; i=i+2)
    {
        wait(); //inserts one clock cycle
        . . . .
       array_0[i] += 5;
        . . . .
    }

Parallel Thread 2 - UNROLL_INSTANCE_2:
for(int i = 1; d < odd_iterations; i=i+2)
    {
        wait(); //inserts one clock cycle
        . . . .
       array_1[i] += 5;
        . . . .
    }
```

Fig. 14. Loop unroll (body) example combined with array splitting.

are then accumulated and the results sent to the DRAM. A large enough FIFO makes sure that the two processes i.e. the path accumulation and arbitration, perform their specific tasks in parallel with minimum synchronization overhead. Several experiments were performed with different sizes of the FIFO channels, after which, it was concluded that a FIFO with sufficient size to accommodate the results of 35 rows of a VGA frame over 128 disparity values provides enough amount of parallelism in order to meet the desired performance, while still conforming onto the on-chip BRAM.

The micro-architectural decisions again focus on the cost cube and path cost computation functions, because these two loops are the computation bottlenecks of the design. We again applied partial unrolling by a factor of two and four respectively. This was combined with array splitting using HLS tool directives. The splitting factor was kept the same as the loop unrolling factor to avoid any race condition among the memory banks. Fig. 14 shows that, if an array is allocated to a BRAM (with separate read and write ports) without partitioning, then scheduling cannot be done. This is because, both the parallel threads would attempt to access the same memory bank simultaneously, thereby causing memory access conflicts. This problem is resolved by splitting the array and mapping it to different memory banks. We could not go beyond a factor of four for unrolling due to the resource constraints of the selected FPGA platform.

E. Forward/backward scan-based architecture

The most refined architecture is quite similar to the architecture proposed by the 3DV project presented in [8]. It heavily relies on the on-chip BRAMs and uses pre-fetching to meet the performance constraints. The path cost calculation (L) step is divided into two scans, i.e. the forward scan and the backward scan. The forward scan consists of the 0°, 45°, 90° and 135° paths, whereas the backward scan consists of the 180°, 225°, 270° and 315° paths, as depicted in Fig. 15. The forward scan starts from the top left pixel, and computes the four path costs for each pixel. The path costs from the forward scan are aggregated and written into the external DDR memory. This is followed by the backward scan starting from the bottom-right pixel. Furthermore, the four paths corresponding to the forward

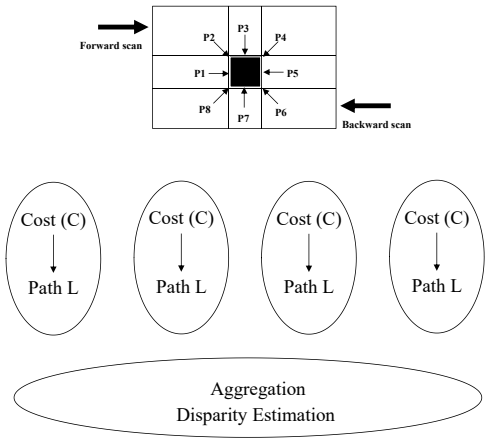

Fig. 15. Illustration of forward/backward scan-based multi-threaded (five) architecture.

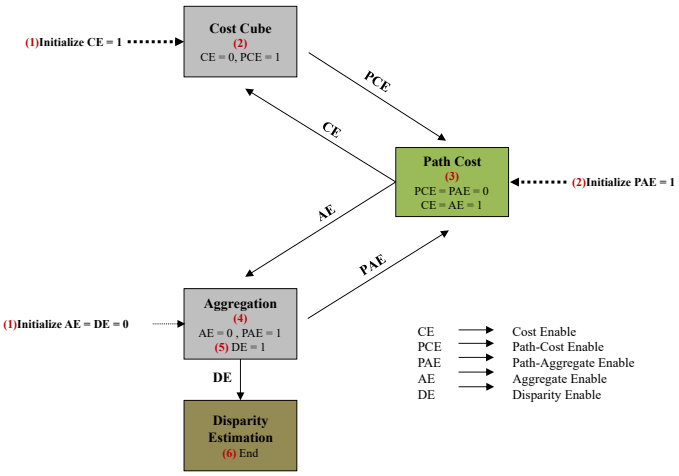

Fig. 17. Top level pipelined among cost cube, path cost and aggregation processes.

scan are added together before being moved to the DDR in order to decrease the bandwidth requirements. The aggregated cost is read back from the DDR and the final aggregated cost is then used to perform the disparity estimation by minimizing all the path costs. We will see shortly that this refinement proved to be good enough to meet the performance of the manual RTL implementation, while keeping the on-chip memory sizes within the bounds of the target platform.

The architectural overview of the parallel path cost calculation of the SGM along with the disparity aggregation and estimation is presented in Fig. 16. For both forward and backward scans, the cost cube results are computed only once and then replicated three more times, so that each path cost process has its own copy. All these intermediate results are stored into arrays which are mapped to on-chip BRAMs. The aggregated result of each pixel is then stored into the external DDR. The backward scan works the same except for an additional read from the DDR. Once all the results of all the paths are aggregated, disparity computation is performed.

It is evident from the architectural block diagram that while path cost aggregation is performed by the path cost and aggregation blocks, the cost cube block remains idle. Thus, we introduce a top-level pipeline between the cost cube and the aggregation processes. Because of this, the cost cube starts computation when the aggregation process is fetching the data from the path cost process, as presented in Fig. 17. The red numbers inside the brackets show the order of the computation steps.

VI. High-Level Synthesis for Semi-Global Matching Algorithm

In the previous sections we discussed the hardware architectures and various micro-architectural choices for each implementation. Now we discuss the experimental setup and the results of various implementations as a result of design space exploration based on HLS. We also compare the results with other relevant state-of-the-art implementations.

A. Design Analysis

The total design effort to undertake the algorithmic as well as architectural refinements with respect to the reference system-level code (executable specification) is compared against the manual RTL implementation. The total line count for the reference code was 917, while the SystemC implementation comprised of approximately 1238 lines (800 for HW, 438 for SW). The number of reused code lines in SystemC was 700 (HW 35%, SW 65%). Thus the estimated additional coding effort for SystemC-based HLS implementation was 43% (measured by counting the total number of code lines in SystemC implementation), with a reuse factor of 76% (of the number of lines in the reference code). It is worth mentioning that the RTL implementations obtained from the automated HLS flow contained roughly 10000 code lines (without the testbench), which also shows the significance of adopting HLS for designs with moderate complexity.

The SGM unit needs to attain a frame rate of 30 fps for VGA images with 128 px disparity range. As mentioned above, the less critical steps of preprocessing and rectification are executed in SW, while the cost cube (C) generation, the path cost accumulation (L) and the disparity estimation are executed in HW.

C-to-Silicon (CTOS) version 13.20 from Cadence Design Systems has been used as an HLS tool in this activity. The SystemC test bench has been used both for the functional verification of the high-level implementation of the algorithm as well as for performance analysis of the corresponding RTL implementation.

B. Design Space Exploration

The main goal of this work was to obtain an HLS-based hardware implementation of SGM, that is comparable to a highly efficient (and hence very rigid) manual RTL implementation that was previously developed by our group [8]. Moreover, we also wanted to perform design space exploration by making several considerably different micro-architectural

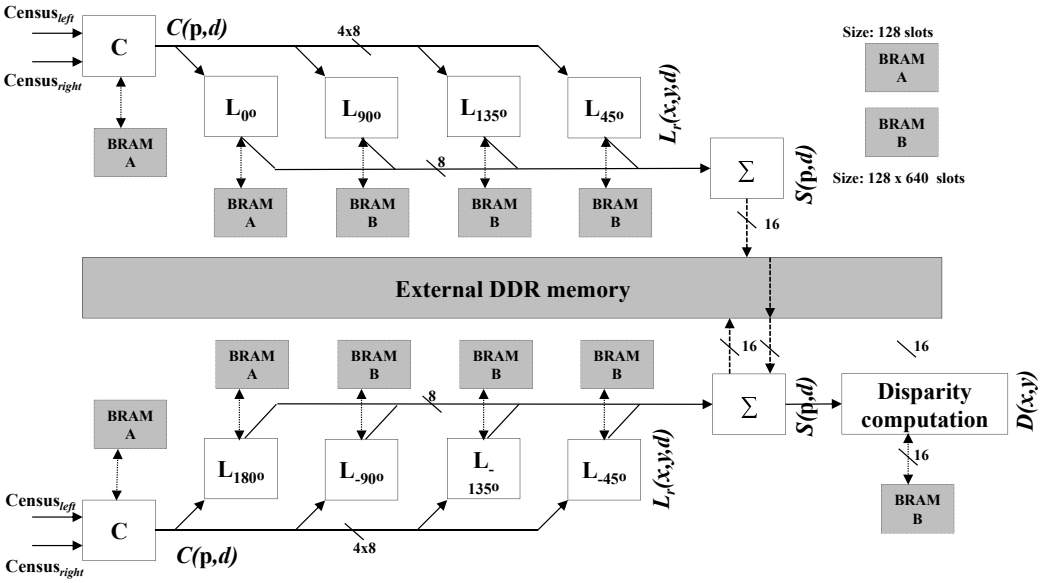

Fig. 16. Architectural overview of the parallel path cost calculation of the semi-global matching.

decisions yielding different implementations of a single design, fitting into a wide range of cost (FPGA resources) and performance curve. Before comparing the implementations as a result of DSE, we present a performance comparison between the manual RTL system (3DV) and other publicly available SGM implementations in Table II. The solution presented in [12] exhibits a better processing time, however it runs on a hardware device with considerably higher cost and power consumption [8]. The implementation proposed in [37] is a scalable architecture with better processing time. However, it takes into account only the path costs in four orientations, which requires intensive post processing steps to improve the image quality, because the disparity values with four path accumulation produce poor resolution. Our selected reference [8] has far better results in terms of execution time than the implementations presented in [40], [41].

Table III presents the area and performance results in terms of cycles per pixels for the three different architectures along with the results of manual RTL implementation. For HLS, all arrays having more than one read or write request at the same time are automatically mapped to distributed RAM which is formed using logic slices. It is worth mentioning that the LUT logic count of all the implementations were within the allowed limits for the Zynq 7020. Also the performance figures were even better than the manual RTL implementation for the architecture based on FIFOs. However, due to the large FIFOs, the BRAM count exceeded the available resources, which motivated us to switch to the scan-based architecture in order to meet the resource constraint of the ZED board.

From Table III, it can be seen that the area is split into four incomparable aspects (LUTs, BRAMs, Distributed RAMs, DSPs). It is impossible thus to plot the area vs. CPP plot in 2 dimensions. Therefore, for better visual comparison, we have made a few simplifications. We have merged the area counts of different resources into a single logic slice figure, based on the information mentioned in the Xilinx 7 series DSP48E1 slice user guide [42]. This implies that the vertical height of a single DSP slice contains 20 LUTs, 36Kb RAM, and 2x18Kb RAM, vertically. It must be noted that the datasheet only provided the vertical height ratios of these resources. So we assumed that the horizontal widths of these resources were the same in order to simplify things. With this simplification, we obtained the area in terms of a single resource i.e. slices, as shown in Table IV.

The plot of design performance in terms of CPP versus area resource in terms of slices is depicted in Fig. 18. The points marked by green triangles are the Pareto-optimal points both in terms of resource utilization as well as performance, obtained from design space exploration. The explored macro-architectural space from a single model spans a 10X range in terms of performance and a 3X range in terms of area. The micro-architectural space for the most efficient model spans a 3X performance range and 2X area range.

VII. Conclusion

This article addresses some of the challenges posed by high-level synthesis tools while trying to improve the QoR for hardware implementations from a system-level behavioral model described at a high abstraction level. This research activity shows how HLS can be used to get several considerably different RTL implementations of a design by specifying different micro-architectural choices. This shall firstly reduce the design time and effort for achieving various performance and cost targets, and secondly, it shall improve the quality of results for a given target by allowing a much wider design space exploration as compared to what can be explored with the manual RTL design. These savings were exhibited by considering a Stereo Vision System example taken from the automotive domain. Several modifications were made to the reference software code before it went through high-level synthesis. These transformations included algorithmic modifications along with wrapping of the C code using SystemC,

TABLE II
COMPARISON OF VARIOUS STEREO RECONSTRUCTION HW IMPLEMENTATIONS

Implementation reference	Choice of HW Platform	Algorithm choice	Image Size [*px*]	Time [*ms*]
Gehrig ECVW10 [40]	Intel® Core™ i7 975 EX@3.3GHz	CT + SGM(8) +MF + L/R	640 x320@128	224
Hirschmller ISVC10 [41]	NVIDIA® GeForce™ 8800 Ultra	HMI +SGM(8) +MF +L/R	640 x 480@128	238
Banz ICCV11 [12]	NVIDIA® Tesla C2050	RT + SGM(8) +MF	640 x 480@128	16
Banz SAMOS10 [37]	Xilinx® Virtex-5 LX 220T-1	RT + SGM(4) + L/R +MF	640 x 480@128	9.7
3DV [8]	Xilinx® Zynq™ 7020	CT + SGM(8) + 2^{nd}min	640 x 480@128	33

Current SGM implementations overview; In parentheses, aggregation paths count. Various cost functions have been utilized i.e. the census transform represented by CT, rank transform given by RT, hierarchical mutual information given by HMI, and zero-mean sum of absolute differences represented by ZSAD. L/R stands for the left-right consistency check, MF stands for median filtering and 2^{nd}min is the minimum vs 2^{nd} minimum ratio check.

TABLE III
DESIGN SPACE EXPLORATION OF SGM ALGORITHM VS. MANUAL RTL IMPLEMENTATION

Refinement	Implementation	Logic LUTs	BRAMs	Distributed RAMs	DSPs	Cycles Per Pixels (CPP)
Sequential Code	No Unroll	14844	-	22	6	83
	Full Unroll	26273	-	40	6	62
FIFO-based Code	No unroll	20504	1542	67	6	21
	Unroll 2x	26373	1187	120	6	10
	Unroll 4x	35663	983	150	6	7
Forward/Backward scan-based Code	No unroll	20055	201	56	6	26
	Unroll 2x	23106	190	97	6	14
	Unroll 4x	27072	186	104	6	10
	Unroll 8x	33584	188	155	6	8
Manual RTL (3DV)	Manual RTL	23600	189	-	48	8

TABLE IV
SGM IMPLEMENTATION RESULTS W.R.T. AREA IN TERMS OF LOGIC SLICES VS. CPP

Refinement	Implementation	Logic Slices	Cycles Per Pixels (CPP)
Sequential Code	No Unroll	764	83
	Full Unroll	1470	62
FIFO-based Code	No unroll	1819	21
	Unroll 2x	1949	10
	Unroll 4x	2317	7
Forward/Backward scan-based Code	No unroll	1123	24
	Unroll 2x	1280	14
	Unroll 4x	1478	10
	Unroll 8x	1818	8
Manual RTL (3DV)	Manual RTL	1322	8

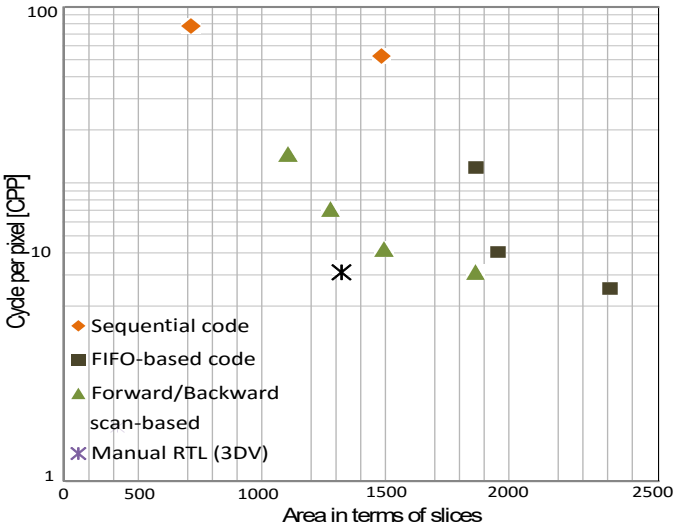

Fig. 18. Performance vs Area Curve for various micro-architectural choices.

mostly aiming to increase the explicit parallelism, so that the HLS tool could more easily exploit it. This was followed by some architectural refinements which mainly dealt with manual address analysis and loop unfolding to assist and improve the results of automated micro-architectural choices. We were eventually able to obtain HLS-based implementations that were comparable to the performance of the equivalent manual RTL design. This hence answered affirmatively the question that was posed in the title of this article i.e. the juice indeed is worth the squeeze. As a future extension of this work, it would be interesting to use the findings of this research activity to try and develop a methodology which can automatically perform or at least advise the transformations that were made manually here to obtain high performance RTL via high-level synthesis.

REFERENCES

[1] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14. [Online]. Available: http://dx.doi.org/10.1109/ISSCC.2014.6757323

[2] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 50.

[3] S. Ravi and M. Joseph, "High-level test synthesis: A survey from synthesis process flow perspective," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 4, p. 38, 2014.

[4] J. Cong, "From design to design automation," in *Proceedings of the 2014 on International symposium on physical design*. ACM, 2014, pp. 121–126.

[5] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*. Springer, 2014, pp. 483–492.

[6] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 641–646.

[7] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore socs," in *Proceedings of the 39th annual Design Automation Conference*. ACM, 2002, pp. 789–794.

[8] G. Camellini, M. Felisa, P. Medici, P. Zani, F. Gregoretti, C. Passerone, and R. Passerone, "3dvan embedded, dense stereovision-based depth mapping system," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*. IEEE, 2014, pp. 1435–1440.

[9] A. Qamar, C. Passerone, L. Lavagno, and F. Gregoretti, "Design space exploration of a stereo vision system using high-level synthesis," in *MELECON 2014-2014 17th IEEE Mediterranean Electrotechnical Conference*. IEEE, 2014, pp. 500–504.

[10] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[11] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and vision computing*, vol. 21, no. 11, pp. 977–1000, 2003.

[12] C. Banz, H. Blume, and P. Pirsch, "Real-time semi-global matching disparity estimation on the gpu," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*. IEEE, 2011, pp. 514–521.

[13] A. Qamar, F. B. Muslim, and L. Lavagno, "Analysis and implementation of the semi-global matching 3d vision algorithm using code transformations and high-level synthesis," in *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. IEEE, 2015, pp. 1–5.

[14] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 243–260, 2004.

[15] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Drdu: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 2, p. 15, 2007.

[16] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching scheme for scratchpad memory," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 960–965.

[17] J. Cong, P. Zhang, and Y. Zou, "Combined loop transformation and hierarchy allocation for data reuse optimization," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2011, pp. 185–192.

[18] Xilinx, *Vivado Design Suite User Guide High-Level Synthesis*, Xilinx.

[19] Cadence, *Cadence C-to-Silicon Compiler User Guide*, Cadence.

[20] Calypto, *Catapult Synthesis Process, Concept and Reference Manual*, Calypto.

[21] "Synphony c compiler," https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx, 2016, [Online;accessed 16-August-2016].

[22] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An integrated and automated memory optimization flow for fpga behavioral synthesis," in *17th Asia and South Pacific Design Automation Conference*. IEEE, 2012, pp. 257–262.

[23] T. Kim and J. Kim, "Integration of code scheduling, memory allocation, and array binding for memory-access optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 142–151, 2007.

[24] Y. Tatsumi and H. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," *Electronics Letters*, vol. 35, no. 25, pp. 2185–2187, 1999.

[25] W. K. Ho and S. J. Wilton, "Logical-to-physical memory mapping for fpgas with dual-port embedded arrays," in *International Workshop on Field Programmable Logic and Applications*. Springer, 1999, pp. 111–123.

[26] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 2, pp. 96–105, 2002.

[27] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 4, p. 61, 2008.

[28] Y. Ben-Asher and N. Rotem, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 155–162.

[29] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 2, p. 15, 2011.

[30] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. Cheung, "Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: a geometric programming framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 305–315, 2009.

[31] Q. Liu, T. Todman, and W. Luk, "Combining optimizations in automated low power design," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 1791–1796.

[32] F. Balasa, H. Zhu, and I. I. Luican, "Computation of storage requirements for multi-dimensional signal processing applications," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 15, no. 4, pp. 447–460, 2007.

[33] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1233–1238.

[34] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2012, pp. 488–495.

[35] Y. Ben-Asher and N. Rotem, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 155–162.

[36] S. K. Gehrig, F. Eberli, and T. Meyer, "A real-time low-power stereo vision engine using semi-global matching," in *International Conference on Computer Vision Systems*. Springer, 2009, pp. 134–143.

[37] C. Banz, S. Hesselbarth, H. Flatt, H. Blume, and P. Pirsch, "Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*. IEEE, 2010, pp. 93–101.

[38] O. S. Initiative, *SystemC 2.0.1 Language Reference Manual*, Open SystemC Initiative.

[39] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides, "Area implications of memory partitioning for high-level synthesis on fpgas," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[40] S. K. Gehrig and C. Rabe, "Real-time semi-global matching on the cpu," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition-Workshops*. IEEE, 2010, pp. 85–92.

[41] I. Ernst and H. Hirschmüller, "Mutual information based semi-global stereo matching on the gpu," in *International Symposium on Visual Computing*. Springer, 2008, pp. 228–239.

[42] Xilinx, *7 Series DSP48E1 Slice User Guide*, Xilinx.